2

COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY STANFORD, CA 94305-4055

Microsupercomputers: Design and Implementation

AD-A233 155

DTIC FILE COPY

Stanford University
Computer Systems Laboratory

Semi-Annual Technical Report

Defense Advanced Research Projects Agency

For the period of October 1990 - March 1991

Contract Number: N00014-87-K-0828

DTIC ELECTE MAR 27 1991 S D D

Principal Investigator
John L. Hennessy

Associate Investigator
Mark A. Horowitz

DISTRIBUTION STATEMENT A
Approved for public release
Distribution [illegible]

91 [illegible]

Semi-Annual Technical Progress Report

October 1990- March 1991

Contract No. N00014-87-K-0828

Order No. 1133

R & T Project Code: 4331685

This work is supported by the Defense Advanced Research Projects Agency and the Office of Naval Research.

The views and conclusions contained in this document are those of the authors and should not [illegible] interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Table of Contents

1. Executive Summary, Goals and Accomplishments 1
2. Technical Progress 3
   - 2.1 Parallel Processor Architecture 3
     - 2.1.1 The DASH Hardware 3
     - 2.1.2 The DASH Operating System 4
     - 2.1.3 Basic Architectural Studies 5
     - 2.1.4 Simulation and Performance Debugging Tools 7
   - 2.2. Parallel Software 8
     - 2.2.1 Parallel Applications Studies 8
     - 2.2.2 Data Dependence Analysis 9
     - 2.2.3 Automatic Blocking 10
     - 2.2.4 Jade Research 11
   - 2.3. Uniprocessor Architecture 12
     - 2.3.1 Super-Scalar Computers 12
   - 2.4. Computer-Aided Design 12
     - 2.4.1 Simulation 12
3. Publications, Presentations, Reports 14
4. Project Staff 17

Accession For

NTIS CRA&I	✓
DTIC TAB	
Unannounced	
Justification	

By per AD-A 228744
Distribution /

Availability Codes

Dist	Avail and / or Special
A-1	

1. Executive Summary, Goals and Accomplishments

DASH Parallel Processor: Our primary focus over the last six months has been the construction of the DASH prototype. We received the fabricated boards that implement the directory-based cache coherence last October. Since then we have been spending most of the time on hardware and software debugging. At the time of this writing, we have two 8 processor (2-cluster) prototypes mostly debugged and working. The prototypes boot UNIX and we have been able to run applications on them. We plan to put together a 16 processor (4-cluster) system later this month.

Operating Systems for DASH: The DASH operating system kernel is currently running on a 2-cluster dash machine, supporting full intercluster memory access, cached dash locks, a master cluster based file system, master cluster swapping, and cluster and processor attachment. Currently, file system and swapping system are being expanded to support transparent multicluster access. Research is being done on operating system support for memory management and scheduling.

Multiprocessor Architectural Studies: We are continuing to study techniques for coping with the large latency of memory accesses in multiprocessors. Evaluating the techniques of hardware coherent caches, relaxed memory consistency, software-controlled prefetching, and multiple-context processors, we show that factors of 4 to 7 improvement in performance can be obtained on modest-sized machines. We have also developed two novel techniques that can significantly improve the performance of sequential consistency on dynamically scheduled processors.

Efficient Simulation of Multiprocessors: We have continued development of Tango, our multiprocessor simulation system. We have extended it to work with light-weight threads and to run on multiprocessors. Our new Tango system performs fully-ordered simulations about 25 times faster than our old version on a uniprocessor. We have also developed two new tools that aid in performance debugging of parallel programs.

Parallel Applications: We have put together a suite of realistic parallel applications (called SPLASH) to provide to the parallel processing community. We hope that a coherent suite of good, realistic applications will allow consistent and comparable architectural evaluations to be performed.

Compiler Management of Memory Locality: Blocking is an important optimization directed to reduce the memory bottleneck found in most computer systems. It is the base optimization technique used in the LAPACK library. We have been successful in applying our automatic blocking algorithm to real codes such as matrix multiplication, a successive over-relaxation (SOR) code, LU factorization without pivoting and Givens QR factorization. Performance evaluation reveals that blocking can improve uniprocessor workstations by a factor of 3-4; its impact on multiprocessor is even more significant as it reduces memory contention, permitting a near-linear speed-up on multiprocessor systems.

Jade: We have made significant progress in the development of Jade, a language for exploiting coarse-grain parallelism. Jade simplifies programming by providing the illusions of sequential execution and a single address space, and it supports portability by hiding the management of the hardware from the programmer. We have pushed the fronts of both programmability and portability by studying the expression of new applications in Jade, and implementing the language on different shared address space machine platforms.

Parallel Simulation: The research on integration of existing simulators into a common simulation paradigm to facilitate a parallel multi-level mixed-mode simulator during this period was concentrated mainly on the development of a prototype on an Intel iPSC/860 message-passing machine. An earlier prototype of a distributed multi-level mixed-mode simulator on conventional workstations has also been tested during this period.

2. Technical Progress

2.1 Parallel Processor Architecture

2.1.1 The DASH Hardware

Since our last report, we have made considerable progress towards completing the 16 processor DASH prototype. The reply controller board was received from fabrication in early October. Within a week after its receipt, we had a single cluster system on which we were able to run diagnostics. After a few problems involving misunderstandings of the underlying Silicon Graphics hardware were fixed, we were able to connect two clusters together. A number of diagnostics have been run over the two cluster system, extensively testing the basic coherence protocol. Currently, we have a two-cluster DASH system set up at Silicon Graphics, with a second two-cluster machine set up at Stanford. The system at Silicon Graphics is used for hardware debug, while the system at Stanford is used for operating system development and hardware performance monitoring. We plan to integrate the two systems into a single 16 processor system by the end of March.

The debugging effort of the DASH hardware has made steady progress.The extensive simulation of the directory-controller and reply-controller boards has paid off well, as the basic data path of both boards has remained unmodified. In addition to the test vectors generated by the functional simulator, vectors corresponding to cases not modelled in the simulator were generated by hand. A few bugs were discovered in the logic that handled these cases, as the simulation coverage there was less complete. An example of an error encountered involved backdoor access of the Remote Access Cache, which was getting improper data due to a control signal not being held valid during the backdoor cycle. However, no **major** problems with the coherence protocol have been found. Most of the modifications to the protocol ROM have resulted from either our own omissions of protocol actions for a couple of states or a miscommunication of state between the reply-controller and directory-controller boards.

The network, which was not modelled in the DASH simulation, proved to be more of a problem. First, crosstalk between adjacent signals on the cable connecting the clusters was causing bits to be corrupted. Special cables in which all signals are separated by a ground, as opposed to a cable containing a single ground plane under the signals has solved the crosstalk problem. More recently, diagnostics written specifically to stress the network uncovered a second problem. Under heavy loading, duplicate network flits were appearing at the output of the mesh routing chips. The network receive logic caught these packets, which had the wrong length, so detection of the problem was relatively straightforward. However, isolation of the problem proved to be more difficult. A good portion of this difficulty was due to the asynchronous nature of the mesh routing chips.

Changes in the interface logic to the mesh routing chips would create subtle changes in the timing of the network handshake signals. These changes would in turn alter the frequency with which the problem occurred. After many weeks of debugging, a connection between timing of the packets entering and leaving the mesh routing chip and the error was observed.

Testing of the routing chip in isolation confirmed the relationship. If the request signal for a packet entering the chip was *slowly* rising at the same time as the request signal for the packet leaving the chip was *quickly* falling, a glitch on the rising incoming request signal would be generated. This glitch got interpreted as a pair of requests inside the chip, and the extra flit was generated. On the mesh routing chip, the pins for the request in and request out signals are physically adjacent, allowing this signal coupling to occur. As a short term solution, the timing of the network requests has been slowed to avoid the simultaneous switching of the requests in and out of the mesh routing chip. This solution has solved the problem in our two-cluster system, however, as a longer term solution we plan on speeding up the edge rates of the requests between mesh routing chips. This will be done by routing the request signal from the cable to a buffer and then to the mesh routing chip, instead of directly from the cable to the chip. Since the request in from the cable will now rise faster, the coupling with the request out of the mesh routing chip will be unable to induce the glitch.

The DASH hardware debugging continues, though we can already boot UNIX and run parallel programs. A diagnostics shell provides a regression suite to check new board changes, and the DASH protocol verifier (DPV) has been ported to the DASH hardware. Hardware problems and software problems with DPV itself prevented its use as a debugging tool until recently. Both the software and hardware problems have been solved, and we plan on using DPV to complement the diagnostics in the DASH hardware test effort.

2.1.2 The DASH Operating System

The DASH Operating System kernel is currently running on a 2-cluster DASH machine, supporting full intercluster memory access, cached DASH locks, a master cluster based file system, master cluster swapping, and cluster and processor attachment. This operating system version has been tested by executing a series of parallel applications as well as IO intensive parallel compilations. To facilitate hardware and software testing and debugging, we have implemented a multithreaded diagnostics support library that allows running threads on different processors directly on the bare hardware, without the assistance of the operating system.

We are working on several operating system enhancements for DASH. A virtual page cluster attachment mechanism is being implemented to support efficient NUMA placement and replacement policies. The file system and swapping system are being expanded to support transparent multicluster access. We have also been doing research on scheduling issues for machines like DASH, and we have developed a novel two-level scheduler that offers high performance by combining the approaches of *process control* and *processor partitioning* [1], [2].

The process control approach is based on the principle that to maximize performance, a parallel application must dynamically match the number of runnable processes associated with it to the effective number of processors available to it. This avoids the problems arising from oblivious preemption of processes and it allows an application to work at a better operating point on its speedup versus processors curve. The processor partitioning is necessary for dealing with realistic multiprogramming environments, where both process controlled and non-controlled applications may be present. It also helps improve the cache performance of applications. We have currently implemented this scheduler on the single cluster of a DASH multiprocessor. Our experiments show that process control can improve performance by as much as two-fold when multiple applications are run simultaneously. We expect to see even more advantage on the multi-cluster DASH, where processor partitioning may be used to restrict an application to one or more clusters when the system is heavily loaded, while still allowing full use of the machine when there is less load.

2.1.3 Basic Architectural Studies

Techniques that can cope with the large latency of memory accesses are essential for achieving high processor utilization in scalable shared-memory multiprocessors. We considered four important architectural techniques that address the latency problem, namely (i) hardware coherent caches, (ii) relaxed memory consistency, (iii) software-controlled prefetching, and (iv) multiple-context processors. While some data has been available in the past regarding the benefits of the individual techniques [3], [4], no study evaluates all of the techniques within a consistent framework. We have closed this gap by providing a comprehensive study of the benefits of the four techniques, both individually and in combinations, using a consistent set of architectural assumptions [5]. The results have been obtained using detailed simulations of a large-scale shared-memory multiprocessor, and the results show that caching shared data and relaxed consistency uniformly improve performance. The improvements due to prefetching and multiple contexts are sizeable, but are much more application-dependent. Combinations of the various techniques generally attain better performance than each one on its own. The exception is multiple contexts with prefetching, which did not work well together.

Overall, we show that using suitable combinations of the techniques, a factor of 4 to 7 improvement in performance can be obtained.

On the subject of memory consistency, we have been continuing our performance evaluation for processors with non-blocking loads to complement our previous results for blocking loads. We have also concentrated on making it easier for a programmer to use architectures with relaxed models. Our previous research addressed this issue by showing that a release consistent architecture provides sequentially consistent executions for programs that are free of data races. However, the burden of guaranteeing that the program is free of data races remained with the programmer. To aid the programmer further, we have developed a unique architectural feature that determines whether sequential consistency is violated in architecture supporting a relaxed consistency model. For every execution of the program, the technique determines *either* that the execution is sequentially consistent *or* that the program has data races and may result in sequentially inconsistent executions. The above mechanism maintains the high performance associated with relaxed consistency models and can be used during normal executions of the program. If the execution is sequentially consistent, the programmer is assured that the relaxed consistency model did not affect the correctness of that execution. And if it is determined that the program has data races, then the programmer knows that it is possible to get sequentially inconsistent results if that program is executed on architectures supporting relaxed models.

We have also studied more efficient implementations of sequential consistency for programmers who are not willing to deal with the extra complexity introduced by relaxing the consistency models. Previously, it was widely believed that sequential consistency could not be implemented without a high performance penalty. We have proposed two techniques that boost the performance of sequential consistency and allow performance close to that of relaxed models like release consistency [6]. The first technique involves prefetching values for accesses that are delayed due to consistency model constraints. The second technique employs speculative execution to allow the processor to proceed even though the consistency model requires the memory accesses to be delayed. We are currently studying the performance of these techniques.

Another area of ongoing investigation has been to evaluate the implementation and performance trade-offs of limited pointer directories for cache coherence [7], [8]. Limited pointer techniques are important to scale directory-based machines to large processor counts. These directories maintain cache coherence by storing several *pointers* with each main memory block; identifying those caches containing the block. By applying an analytic model of parallel workload behavior (verified against multiprocessor address traces) to the state transition graphs implemented by the directory, we can easily estimate the performance of limited pointers under various large-scale workloads. While we have demonstrated that limited pointer directories show good performance in general, blocks that exhibit a high ratio of read to write references yield performance levels that are suboptimal. We have recently developed a scalable *dynamic pointer allocation* directory that shows good potential for supplying the highest performance possible under all but the most extreme workload conditions. Rather than building a fixed number of pointers per entry into the hardware, this scheme allocates pointers as they are needed from a pool of available pointers. We have identified the differences in the resulting protocol relative to standard limited pointers directory organizations, including the steps taken in exceptional circumstances, such as running short of available pointers. We have also detailed one possible implementation approach and examined some potential performance optimizations.

2.1.4 Simulation and Performance Debugging Tools

We have continued development of our multiprocessor simulation system, Tango. While Tango is faster than our previous tools and has enabled many useful studies, large simulations (involving hundreds of processors, complex memory hierarchies, and large applications) are often too time-consuming to be practical with Tango. Our goal is to speed simulations in three ways: (i) we have developed a successor to Tango that uses light-weight threads instead of full-weight processes; (ii) we are extending light-weight Tango to run effectively on multiprocessors and ultimately on DASH; and (iii) we are investigating the usefulness of simpler memory simulators for studies of complex memory hierarchies. This is critical since simulation of the network and memory system dominates total simulation time in many experiments. We believe that memory simulations can be made less expensive by carefully analyzing the level of detail required in simulation models and judiciously trading-off accuracy for efficiency. The new Tango performs fully-ordered simulations about 35x faster than the old on a uniprocessor.

We are currently working on two tools that aid in performance debugging of programs. Our first tool is called MTOOL [9]. It provides support for performance debugging of parallel programs. The current implementation is for programs written with the ANL macros running on MIPS-based multiprocessors (including DASH). In just more than twice the time for a single execution of a parallel C or Fortran application on a given input, MTOOL will develop hierarchical information on the distribution of execution time for the program on that input broken down into:

1. CPU execution time
2. Overhead in accessing the memory hierarchy
3. Idle time waiting on synchronization (locks and barriers)
4. Parallel overhead
5. System Time

The first four classes of execution time may be viewed for the whole program, per process, per procedure, and loop level. MTOOL constructs its program profile using a combination of timer calls and execution time estimates based on basic block counts. The basic block counting is accomplished with at a minimal perturbation to the program. The basic block count information allows MTOOL to construct the CPU execution time numbers, which in conjunction with measured execution times, enable MTOOL to estimate the memory hierarchy overheads.

To understand and remedy performance bottlenecks, users often require information about memory behavior at an even lower level than what MTOOL provides, that is, at the level of individual data structures and procedures. Our second tool provides information such as cache miss rates, memory latencies, and causes of cache misses. It helps the user determine: whether the miss rate is high because of cold start misses, invalidation misses, or replacement misses, which data structures are interfering with each other, and further information about the program execution at this level. By systematizing a process to associate regions of shared memory with high level program names, we can present data to users in terms of data structures and procedures they are familiar with, rather than, for example, in terms of cache blocks. In its current form, this memory characterization tool works with the Tango memory simulator. We intend to further develop the tool by creating a version which uses the DASH hardware performance monitor to collect program events in real-time, rather than through Tango simulations. We consider this tool to be an important component of a complete performance debugging framework which provides the user with information both on where a program's performance bottlenecks are, and why they are occurring.

2.2. Parallel Software

2.2.1 Parallel Applications Studies

Designers of parallel systems are faced with a chicken and egg problem regarding applications software. Few real applications exist to guide their design, and users are unwilling to write new applications for systems that do not exist. The result is that studies done to evaluate system features often base their conclusions on "toy" programs that bear little resemblance to, or are only a part of, the codes people will actually run on these systems. We have put together a suite of realistic parallel applications (called SPLASH) to provide to the parallel processing community [10]. We hope that a coherent suite of good, real applications will allow consistent and comparable evaluations to be performed. We have also put together a detailed documentation of the applications and their characteristics, providing a common reference point for the writers and readers of evaluation studies. The applications and documentation are likely to be released within the next few weeks. The programs, many of which have been developed at Stanford, include five complete applications (an ocean simulation, a N-body molecular dynamics simulation, a Monte Carlo rarefied hypersonic flow simulation, a global router for VLSI, and a distributed-time circuit simulator) and three basic routines (two graph problems and a sparse Cholesky factorization routine). Three other applications–a finite element program, the Greengard-Rokhlin adaptive algorithm for N-body problems, and a multigrid solver–are currently under development.

We have also been continuing our research on scalable parallelism in some real scientific applications. Dividing the problem into finding parallelism and implementing it for efficient performance, we are taking a quantitative look at the impact of various transformations in enhancing scalable performance. We are trying to learn from this effort what types of issues the programmer must be concerned about if scalable performance is desired, what features are most desirable in parallel tools and environments, and what the implications are for the design of scalable architectures. Some results on finding parallelism have already been reported; the recent emphasis has been on the tradeoff between data locality (for which results will soon be available) and load-balancing, as well as on detailed interactions with a high-latency hierarchical memory system.

One parallel application that we have been studying in great detail is sparse Cholesky factorization. We have been considering alternative strategies for distributing the sparse matrix among the processors to increase concurrency and reduce communication. In particular, we have been looking at methods that distribute rectangular submatrices of the sparse matrix among the processors, instead of the more traditional approach of distributing entire sets of columns (supernodes). The asymptotic advantages of submatrix-oriented methods are easily demonstrated through simple growth-rate calculations. However, preliminary results indicate that these advantages do not come into play for matrix sizes and multiprocessor configurations that we see now or expect to see in the near future. Our immediate goal is to determine at what problem and machine sizes the asymptotic advantages of such techniques will become important.

2.2.2 Data Dependence Analysis

In previous work, we had developed a data dependence analysis system which allowed us to give exact results efficiently in all cases we have seen in practice. Our old algorithm required all references and bounds to be linear functions of the induction variables. In the past half year, we have successfully extended our system to handle unknown symbolic terms, without loss of efficiency. It was commonly believed that symbolic testing is very important in data dependence analysis. Our empirical results indicate that symbolic testing expands the number of unique dependence analysis tests by only about 10%, a much lower number than expected. This can be attributed to the fact that, besides being a parallelizer, our compiler is also an optimizing scalar compiler. It employs aggressive optimizations, including constant propagation, induction variable detection and forward substitution, which tend to reduce the number of symbolic terms in array references. This demonstrates the importance of integrating the scalar and parallelizing compilers into one system.

We have also compared the quality of our algorithm to more standard methods: the GCD test and Banerjee's test. The programs we used for comparison are the Perfect Club Benchmarks, a set of 13 scientific Fortran programs ranging in size from 500 to 18,000 lines. Our algorithm is able to detect 16% more independent references, resulting in 22% fewer direction vectors than these algorithms. This can potentially lead to a much greater degree of exploitable parallelism [11].

2.2.3 Automatic Blocking

We have developed a mathematical formulation of the data locality optimization problem. We introduce two concepts: the *reuse vector space* and *reuse factors* characterize the potential reuse in an algorithm. The *localized vector space*, created by blocking or tiling, is the space in which reuse can be exploited. The locality value is derived from the intersection of these two vector spaces. The data locality optimization problem is to find a unimodular and tiling transform that optimizes the locality value [12].

This analysis yields two important results. First, all transformed code with the same localized vector space belong to the same equivalence class. This observation significantly prunes the search for the optimal transformation. Second, the locality value is much more sensitive to the dimensionality of reuse exploited than to block sizes; this leads to the approach of choosing the loops to block innermost before choosing the block size.

Unlike the stepwise transformation approach used in existing compilers, our loop transformer uses a *compound* transformation that combines permutation, skewing and reversals directly. This is made possible by our theory that unifies loop transformations as unimodular matrix transformations on dependence vectors with either direction or distance components. The algorithm extracts the dependence vectors, determines the best compound transform, then transforms the loops and their loop bounds once and for all. The elegance of the theory significantly simplifies the implementation of the algorithm. Programs the algorithm can block successfully include matrix multiplication, successive over-relaxation (SOR), LU factorization without pivoting, and Givens QR factorization.

Evaluation of the performance on a SGI 8-processor machine indicates that blocking is very important. Blocking improves the performance on a single processor by a factor of 2.75. The effect of tiling on multiple processors is even more significant since it not only reduces the average data access latency but also the memory bandwidth required. Without blocking, contention over the memory bus limits the speedup to about 4.5 times. Blocking permits speedups of over seven for eight processors, achieving an impressive speed of 64 MFLOPS.

2.2.4 Jade Research

Given the current proliferation of parallel architectures, programmers should not have to rewrite their parallel applications for each different machine. Jade provides a high-level concurrency model that insulates the programmer from the low-level machine specific details. To demonstrate this portability, we have implemented Jade on several different platforms, including the Encore Multimax, the Silicon Graphics IRIS 4D/240 and the Stanford Tango simulator running on a sequential machine. The programmer can concentrate on providing the high-level information necessary to parallelize the application, while the Jade system maps the computation efficiently onto the hardware. Given our experience porting Jade to these different platforms, we expect to port Jade easily to the Stanford DASH Multiprocessor [13].

We have tested our design of the Jade language by implementing several large applications. These applications illustrate how Jade supports both irregular dynamic dependencies, as well as structured static dependence patterns. One of these applications is sparse Cholesky factorization–the computational bottleneck of such important computations as linear programming, device simulation and finite-element analysis. This application parallelizes well in Jade even though the parallelism is highly dependent on the input data. The UNIX make utility, another application with irregular parallelism, is also easily parallelized with Jade. We have also parallelized a number of additional applications which have more regular concurrency patterns. These include LocusRoute, a VLSI circuit router, and the Perfect Club benchmark MDG.

We have investigated various performance enhancements for the Jade system, based on our work on Jade applications and the demands of modern multiprocessor architectures. One of the most critical issues in achieving high performance on these modern multiprocessors is optimizing for data locality. We are currently investigating a task placement heuristic for Jade that attempts to minimize the amount of data that must be transferred between processors. This heuristic executes tasks accessing the same data on the same processor.

2.3 Uniprocessor Architecture

2.3.1 Super-Scalar Computers

For the past six months, we concentrated on the hardware and software aspects of the new architecture proposed for superscalar processors. This architecture combines the advantages of dynamic and static scheduling techniques, while minimizing the short-comings of each, to increase the performance of non-numerical applications. This is accomplished through a technique called boosting which allows the compiler to use sophisticated scheduling techniques on both sequential and speculatively-executed instructions. Efficient speculative execution is supported by shadow structures in the hardware that commit boosted state on correct branch prediction and squash boosted state on incorrect predictions.

Currently, a group of five graduate students are looking in depth at the issues involved in building a VLSI processor to support boosting and superscalar execution. We have a logic-level simulator running, and are in the process of determining exactly what hardware is needed to implement each of the functions in the simulator. We are also beginning work on an instruction-level simulator to allow us to verify the results of our logic-level simulator.

Work continues on the scheduling passes for our compiler system that will generate code for our architecture. A significant amount of time has been spent trying to allow one to easily change the underlying instruction set without having to make major changes to the scheduler or to the scheduling algorithms. To aid in the development of the hardware, we have also coded a simple assembler to allow us to create programs for the hardware simulator.

2.4 Computer-Aided Design

2.4.1 Simulation

Parallel simulation and CAD applications have been proposed to further explore the potentials of the DASH multiprocessor machine and of parallel paradigms in general. Parallelism is obtained at a simulator level by decomposing its simulation into smaller blocks and managing the communication and synchronization of these blocks, as described in earlier reports.

The first prototyp [illegible] the parallel multi-level simulator was tested on conventional workstations. The second step is to extend the prototype onto parallel machines. We have successfully ported the simulator onto an Intel N-cube multiprocessor and will eventually, after the DASH multiprocessor machines is stabilized, test the simulator on DASH.

During the last six months, we continued the testing of the first prototype on a network of workstations. The performance gain of using multiple workstations in concurrent simulation was tapered by the communication overhead; still we obtained a speedup factor of 2 or more by running parallel multi-level simulation on five workstations. The correctness of the simulation, and thus the parallel programming of our multi-level mixed-mode simulator, was also verified.

The second prototype was developed on an Intel iPSC/860 message-passing machine and it is currently being tested. The decision to install the second prototype on the iPSC/860 was based on machine availability and a match with our framework of communication mechanism that exchange information through messages. The prototype installation on the iPSC/860 requires a change of localized portions of the simulation kernel that handles the communication between different instances of the simulation nodes. It also requires us to port the simulation programs being integrated (THOR, SPICE, and IRSIM, at the moment) to the Intel iPSC/860 node. The performance gain of using a message-passing multiprocessor for parallel multi-level simulation will be tested and measured.

3. Publications, Presentations, Reports

1. Gupta, A., Tucker, A. and Urushibara, S., The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications, *ACM SIGMETRICS '91*, May, 1991.

2. Tucker, A., Stevens, L. and Gupta, A. "Making Effective Use of Shared-Memory Multiprocessors: The Process Control Approach," October, 1991. Submitted for publication.

3. Gharachorloo, K., Gupta, A. and Hennessy, J. L., Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors, ACM/IEEE, *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Santa Clara, CA. April, 1991.

4. Mowry, T. and Gupta, A. "Tolerating Latency Through Software-Controlled Prefetching in Scalable Shared-Memory Multiprocessors," *Journal of Parallel and Distributed Computing*. June, 1991. To appear.

5. Gupta, A., Hennessy, J., Gharachorloo, K., Mowry, T. and Weber, W.-D., Comparative Evaluation of Latency Reducing and Tolerating Techniques, *18th International Symposium on Computer Architecture*, May, 1991.

6. Gharachorloo, K., Gupta, A. and Hennessy, J., Two Techniques to Enhance the Performance of Memory Consistency Models, *International Conference on Parallel Processing*, 1991.

7. Simoni, R. and Horowitz, M., Dynamic Pointer Allocation for Scalable Cache Coherence Directories, *International Symposium on Shared Memory Multiprocessing*, Tokyo, Japan. April 2-4, 1991.

8. Simoni, R. and Horowitz, M., Modeling the Performance of Limited Pointers Directories for Cache Coherence, *18th International Symposium on Computer Architecture*, May 27-30, 1991.

9. Goldberg, A. and Hennessy, J., MTOOL: A Method for Isolating Memory Bottlenecks in Shared Memory Multiprocessor Programs, *International Conference on Parallel Processing (ICPP)*, 1991.

10. Singh, J. P., Weber, W.-D. and Gupta, A., *SPLASH: Stanford Parallel Applications for Shared-Memory*, Stanford University, Computer Systems Lab, Technical Report Report, 1991. Submitted for publication.

11. Maydan, D. E., Hennessy, J. L. and Lam, M. S., An Efficient Method for Exact Data Dependence Analysis, *ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, June, 1991.

12. Wolf, M. E. and Lam, M. S., A Data Locality Optimizing Algorithm, *ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, June, 1991.

13. Lam, M. and Rinard, M., Coarse-Grain Parallel Programming in Jade, ACM Sigplan, *3rd Symposium on Principles and Practice of Parallel Programming*, April, 1991.

14. Gibbons, P. B. "A Synthesis of Parallel Algorithms." Asynchronous PRAM Algorithms. Reif ed. 1990 Morgan-Kaufmann. San Mateo.

15. Lam, M. "The Software Pipelining Algorithm and Experimental Results," *Transactions on Programming Languages and Systems*. 1990. Submitted.

16. Rothberg, E. and Gupta, A. "Efficient Sparse Matrix Factorization on High-Performance Workstations--Exploiting the Memory Hierarchy," *ACM Transactions on Mathematical Software*. 1991. To appear

17. Rothberg, E. and Gupta, A., *A Comparative Evaluation of Nodal and Supernodal Parallel Sparse Matrix Factorization: Detailed Simulation Results*, Stanford University, Computer Systems Laboratory, Technical Report, CSL-90-416, February, 1990. Also appears as STAN-CS-90-1305 published under the auspices of the Computer Science Department.

18. Singh, J. and Hennessy, J. L. "Parallelizing an Ocean Simulation Program: Experience, Results and Implications," *Journal of Parallel and Distributed Computing*. 1990. Submitted.

19. Berlin, A. and Weise, D. "Compiling Scientific Code using Partial Evaluation," *IEEE Computer*. 23, (9): December, 1990.

20. Rothberg, E. and Gupta, A., Techniques for Improving the Performance of Sparse Matrix Factorization on Multiprocessor Workstations. IEEE Computer Society, *Supercomputing '90*, New York , NY. November, 1990.

21. Chow, F. and Hennessy, J. "The Priority-based Coloring Approach to Register Allocation," *IEEE Transactions on Programming Languages and Systems*. October, 1990.

22. Weise, D. and Ruf. E., *Computing Types During Program Specialization*, Stanford University, Computer Systems Lab, Technical Report Report, CSL-TR-90-441, October, 1990.

23. Acharya, A., Tambe, M. and Gupta, A. "Implementation of Production Systems on Message-Passing Computers," *IEEE Transactions on Parallel and Distributed Systems*. 1991. To appear.

24. Gharachorloo, K. and Gibbons, P., Detecting Violations of Sequential Consistency, *3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, 1991.

25. Gupta, A. and Weber, W.-D. "Cache Invalidation Patterns in Shared-Memory Multiprocessors," *IEEE Transactions on Computers*. 1991. To appear.

26. Saraswat, V., Rinard, M. and Panangaden, P., Determinate Constraint Programming, 1991.

27. Saraswat, V., Rinard, M. and Panangaden, P., A Model for Concurrent Constraint Programming, 1991.

28. Singh, J. P. and Hennessy, J. L., Automatic and Explicit Parallelization of an N-Body Simulation. *IEEE TENCON '91*, 1991.

29. Singh, J. P. and Hennessy, J. L. "Parallelizing the Simulation of Ocean Eddy Currents," *Journal of Parallel and Distributed Computing*. 1991. To appear.

30. Torrellas, J., Lam, M. and Hennessy, J. L. "Measurement, Analysis, and Improvement of the Cache Behavior of Shared Data in Cache Coherent Multiprocessors," *IEEE Transactions on Computers*. 1991. To appear.

31. Lam, M. S., Rothberg, E. E. and Wolf, M. E., The Cache Performance and Optimizations of Blocked Algorithms. *Fourth International Conference on Architectural Support for Programming Languages - ASPLOS -IV*, April, 1991.

32. Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.-D., Gupta, A. and Hennessy, J. L., Overview and Status of the Stanford DASH Multiprocessor, *ISSMM Conference*, Tokyo, Japan. April, 1991.

33. Singh, J. P. and Hennessy, J. L., An Empirical Investigation of the Effectiveness and Limitations of Automatic Parallelization, *International Symposium of Shared Memory Multiprocessing (ISSMM)*, Tokyo, Japan. April, 1991.

34. Wolf, M. E. and Lam, M. S. "A Loop Transformation Theory and Algorithm to Maximize Parallelism," *IEEE Transactions on Parallel and Distributed Systems*. July, 1991. To appear.

35. Wolf, M. and Lam, M., A Loop Transformation Theory and Algorithm to Maximize Parallelism, *Principles of Programming Languages*, January, 1991.

36. Williams, T. E., Analyzing the Latency and Throughput Performance of Self-Timed Pipelines and Rings, *VLSI-91 IFIP Conference*, August, 1991.

37. Williams, T. E. and Horowitz, M. A., A 160nS Division Implementation Using Self-Timing and Symmetric Overlapped Execution, *IEEE Conference on Computer Arithmetic (ARITH-10)*, June, 1991.

4. Project Staff

Faculty:

John Hennessy Principal Investigator	jlh@vsop.stanford.edu	415/725-3712
Mark Horowitz Associate Investigator	horowitz@chroma.stanford.edu	415/725-3707
Anoop Gupta	ag@pepper.stanford.edu	415/725-3716
Monica Lam	lam@k2.stanford.edu	415/725-3714
Daniel Weise	daniel@mojave.stanford.edu	415/725-3711
Teresa Meng	meng@tilden.stanford.edu	415/725-3636
David Dill	dill@amadeus.stanford.edu	415/725-3642

Research Staff:

Charlie Orgish
David Nakahira
Laura Schrager

Graduate Students:

Saman Amarasinghe	Martin Rinard
Jennifer Anderson	Ed Rothberg
Rohit Chandra	Arturo Salz
Tom Chanak	Dan Scales
Helen Davis	Rich Simoni
Andrew Erlichson	JP Singh
Kourosh Gharachorloo	Mike Smith
Aaron Goldberg	Larry Soule
Steve Goldschmidt	Don Stark
Truman Joe	Luis Stevens
Lydia Kavraki	Steve Tjiang
Jim Laudon	Anthony Todesco
Dan Lenoski	Josep Torrellas
John Maneatis	Andrew Tucker
Margaret Martonosi	Wolf Weber
Dror Maydan	Ted Williams
Arul Menezes	Malcolm Wing
Todd Mowry	Drew Wingard
Jason Nieh	Michael Wolf
Karen Pieper	

Comparative Evaluation of Latency Reducing and Tolerating Techniques

Anoop Gupta, John Hennessy,
Kourosh Gharachorloo, Todd Mowry, Wolf-Dietrich Weber

Computer Systems Laboratory
Stanford University, CA 94305

Abstract

Techniques that can cope with the large latency of memory accesses are essential for achieving high processor utilization in scalable shared-memory multiprocessors. In this paper, we consider four prominent architectural techniques that address the latency problem, namely (i) hardware coherent caches, (ii) relaxed memory consistency, (iii) software-controlled prefetching, and (iv) multiple-context processors. While some data has been available in the past regarding the benefits of the individual techniques, no study evaluates all of the techniques within a consistent framework. This paper attempts to fill the above gap by providing a comprehensive study of the benefits of the four techniques, both individually and in combinations, using a consistent set of architectural assumptions. The results in this paper have been obtained using detailed simulations of a large-scale shared-memory multiprocessor. Our results show that caches and relaxed consistency uniformly improve performance. The improvements due to prefetching and multiple contexts are sizeable, but are much more application-dependent. Combinations of the various techniques generally attain better performance than each one on its own. The exception is multiple contexts with prefetching, which did not work well together. Overall, we show that using suitable combinations of the techniques, a factor of 4 to 7 improvement in performance can be obtained.

1 Introduction

Large-scale shared-memory multiprocessors are expected to have remote memory reference latencies of several tens to hundreds of processor cycles [19, 23, 26, 31]. The large latencies arise partly due to the increased physical dimensions of the parallel machine and partly due to the ever increasing clock rates at which the individual processors operate. These large memory latencies can quickly offset any performance gains expected from the use of parallelism. Techniques that can help to reduce or hide these latencies are essential for achieving high processor utilization.

To cope with the large latencies, several different architectural techniques have been proposed. *Coherent caches* [3, 4, 19, 31] allow shared read-write data to be cached and significantly reduce the memory latency seen by the processors. *Relaxed memory consistency models* [1, 6, 9] hide latency by allowing buffering and pipelining of memory references. *Prefetching* techniques [12, 17, 22, 24] hide the latency by bringing data close to the processor before it is actually needed. *Multiple contexts* [3, 13, 14, 27] allow a processor to hide latency by switching from one context to another when a high-latency operation is encountered.

Revised version to appear in 1991
Internat[illegible] S[illegible]sia[illegible] C[illegible]mputer
Architecture

Our primary objective in this paper is to characterize the benefits and costs of these four latency hiding techniques in a systematic and consistent manner. Although one can find papers that focus on the performance of the individual techniques [8, 12, 30], it is not possible to use these papers to perform a comparative evaluation, since frequently the benchmark programs used are different, or the architectural assumptions made are different, or both. We believe that a consistent comparative evaluation is essential to understanding the tradeoffs implicit in the use of the different techniques. Furthermore, since several of the techniques can be provided on the same multiprocessor, the second objective of this paper is to evaluate the interactions and gains from the combined use of the various techniques.

The results presented in this paper are obtained from detailed architectural simulations performed for three parallel applications. The architecture used is based on the Stanford DASH multiprocessor [19], a large-scale shared-memory multiprocessor that provides coherent caches, a relaxed memory consistency model, and support for software-controlled prefetching. The applications we study are a particle-based simulator used in aeronautics (MP3D) [21], an LU-decomposition program (LU), and a digital logic simulation program (PTHOR) [28]. The applications are typical of those that may be found in an engineering environment.

Our results show that the provision of coherent caches leads to significant performance benefits. In fact, for this reason, all remaining experiments in the paper were done assuming that coherent caches are provided. Our studies of the sequential consistency model versus relaxed memory consistency models show that relaxed consistency models uniformly improve performance. Prefetching and multiple-context processors also provide performance improvements, but the magnitude varies considerably depending on the application. Combinations of relaxed consistency with prefetching, or relaxed consistency with multiple contexts work well. Surprisingly, no further gains are achieved when both prefetching and multiple contexts are used. Overall, a suitable combination of the latency reducing/tolerating techniques discussed in this paper boost performance by a factor of 4 to 7 for the applications studied.

The paper is organized as follows. Section 2 describes the architectural assumptions, the benchmark applications, and the simulator used in this study. Simulation results for the performance of each of the techniques are presented in Sections 3–6. Finally, we conclude in Section 7.

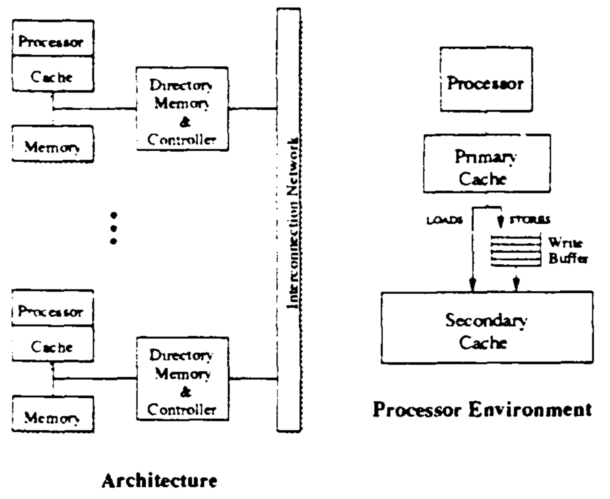

Figure 1: Architecture and processor environment.

Read Operations	
Hit in Primary Cache	1 pclock
Fill from Secondary Cache	14 pclock
Fill from Local Node	26 pclock
Fill from Home Node (Home ≠ Local)	72 pclock
Fill from Remote Node (Remote ≠ Home ≠ Local)	90 pclock
Write Operations	
Owned by Secondary Cache	2 pclock
Owned by Local Node	18 pclock
Owned in Home Node (Home ≠ Local)	64 pclock
Owned in Remote Node (Remote ≠ Home ≠ Local)	82 pclock

Table 1: Latency for various memory system operations in processor clock cycles (1 pclock = 30 ns).

2 Multiprocessor Architecture, Benchmark Applications, and Simulator

To enable meaningful performance comparisons between the different techniques it is necessary to focus on a specific class of multiprocessor architectures. The reason is that the tradeoffs may vary depending on the architecture chosen. For example, the tradeoffs for a small bus-based multiprocessor where broadcast is possible and miss latencies are ten to twenty cycles are quite different from the tradeoffs for a scalable multiprocessor where broadcast is not possible and miss latencies may be a hundred or more cycles. This section presents the architectural assumptions, the benchmark applications, and the simulation environment used to get the performance results.

2.1 Architectural Assumptions

For this study, we have chosen an architecture that resembles the DASH multiprocessor [19], a large-scale cache-coherent machine currently being built at Stanford. Figure 1 shows the high-level organization of the simulated architecture. The architecture consists of several processing nodes connected through a low-latency scalable interconnection network. Physical memory is distributed among the nodes. Cache coherence is maintained using an invalidating, distributed directory-based protocol. For each memory block, the directory keeps track of remote nodes caching it. When a write occurs, point-to-point messages are sent to invalidate remote copies of the block. Acknowledgement messages are used to inform the originating processing node when an invalidation has been completed.

We use the actual parameters from the DASH prototype wherever possible, but have removed some of the limitations that were imposed on the DASH prototype due to design effort constraints. Figure 1 also shows the organization of the processor environment. Each node in the system contains a 33MHz MIPS R3000/R3010 processor connected to a 64 Kbyte write-through primary data cache. The write-through cache enables processors to do single-cycle write operations. The primary data cache interfaces to a 256 Kbyte secondary write-back cache. The interface consists of a read buffer and a write buffer. The write buffer is 16 entries deep. Reads can bypass writes in the write buffer if permitted by the memory consistency model. Both the primary and secondary caches are lockup-free [15], direct-mapped, and use 16 byte lines. The bus bandwidth of the node bus is 133 Mbytes/sec, and the peak network bandwidth is approximately 150 Mbytes/sec into and 150 Mbytes/sec out of each node.[1]

The latency of a memory access in the simulated architecture depends on where in the memory hierarchy the access is serviced. Table 1 shows the latencies for servicing accesses at different levels of the hierarchy, in the absence of contention. The latency shown for writes is the time for retiring the request from the write buffer. This latency is the time for acquiring exclusive ownership of the line, which does not necessarily include the time for receiving acknowledgement messages from invalidations. The following naming convention is used for describing the memory hierarchy. The *local node* is the node that contains the processor originating a given request, while the *home node* is the node that contains the main memory and directory for the given physical memory address. A *remote node* is any other node.

2.2 Benchmark Programs

In this subsection we describe the computational structure of the three benchmark applications used in this paper. This information will be useful in later sections for understanding the performance results. The selected applications are representative of algorithms used in an engineering computing environment. All of the applications are written in C. The Argonne National Laboratory macro package [20] is used to provide synchronization and sharing primitives. Some general statistics for the benchmarks are shown in Table 2.

MP3D [21] is a 3-dimensional particle simulator. It is used to study the pressure and temperature profiles created as an object flies at high speed through the upper atmosphere. The primary data objects in MP3D are the particles (representing the air molecules), and the space cells (representing the physical space, the boundary conditions, and the flying object). The overall computation of MP3D consists of evaluating the positions and velocities of particles over a sequence of time steps. During each time step, the particles are picked up one at a time and moved according to their velocity vectors. If two particles come close to each other, they may undergo a collision based on a probabilistic model. Collisions with the object and the boundaries are also modeled. The simulator is well suited to parallelization because each particle can be treated independently at each time step. The program is parallelized by statically dividing the particles equally among the processors.[2] The main synchronization consists of barriers between each time step. For our experiments we ran MP3D with 10,000 particles, a 14x24x7 space array, and simulated 5 time steps.

LU performs LU-decomposition of dense matrices. The primary

[1]The architectural parameters in this paper differ from those in previous papers [8, 22], and therefore results should not be compared directly.

[2]To minimize cache miss penalties, the particles assigned to a processor are allocated from shared-memory in that processor's node.

Table 2: General statistics for the benchmarks.

Program	Useful Cycles *(K)*	Shared Reads *(K)*	Shared Writes *(K)*	Locks	Barriers	Shared Data Size *(KBytes)*
MP3D	5,774	1170	530	0	448	536
LU	27,861	5543	2727	3184	29	653
PTHOR	19,031	3774	454	75,878	2016	2925

data structure in LU is the matrix being decomposed. Working from left to right, a column is used to modify all columns to its right. Once all columns to the left of a column have modified that column, it can be used to modify the remaining columns. Columns are statically assigned to the processors in an interleaved fashion. Each processor waits until a column has been produced, and then that column is used to modify all columns that the processor owns.[3] Once a processor completes a column, it releases any processors waiting for that column. For our experiments we performed LU-decomposition on a 200x200 matrix.

PTHOR [28] is a parallel logic simulator based on the Chandy-Misra simulation algorithm. Unlike centralized-time algorithms, this algorithm does not rely on a single global time during simulation. The primary data structures associated with the simulator are the logic elements (e.g., AND-gates, flip-flops), the nets (wires linking the elements), and the task queues which contain activated elements. Each processor executes the following loop. It removes an activated element from one of its task queues and determines the changes on that element's outputs. It then looks up the net data structure to determine which elements are affected by the output change and schedules the newly activated elements on to task queues. In the case that a processor runs out of tasks, it spins on the task queues until a new task is scheduled. This time shows up as busy time in our experiments, even though it should rightfully be counted as synchronization time. This fact leads to variations in busy time from experiment to experiment, even though the amount of useful work being done remains approximately the same. For our experiments we simulated five clock cycles of a small RISC processor consisting of the equivalent of 11,000 two-input gates.

2.3 Simulation Environment

An event-driven simulator is used to simulate the major components of the architecture at the behavioral level. For example, the caches, the cache coherence protocol, the contention and arbitration for buses, are all modeled in detail. The simulations are based on a 16 processor configuration. We do not go beyond 16 processors since the concurrency requirements are very large for multiple context simulations. For example, when modeling 4 hardware contexts per processor, 16 processors require the application to support 64 concurrent processes. Some of our existing applications do not scale well beyond 64 threads. The architecture simulator is tightly coupled to the Tango reference generator [10] to assure a correct interleaving of accesses. For example, a process doing a read operation is blocked until that read completes, where the latency of the read is determined by the architecture simulator. Unless specific directives are given by an application, main memory is distributed uniformly across all nodes using a round-robin page allocation scheme.

We now come to a difficult methodological problem that shows up when simulating large multiprocessors. Given that detailed simulators are enormously slower than the real machines being simulated, one can only afford to simulate much smaller problems/applications than those that would be run on the real machine. The question arises of how to scale the machine parameters so as to get realistic performance estimates. For example, consider the MP3D application. In real life, the application is run with enough particles to fill the complete main memory of a machine. Since at each time step in the application all particles are moved (i.e., the complete memory is swept through), the caches are expected to miss on each particle. Had we retained the 64 Kbyte primary and 256 Kbyte secondary caches in the simulator, then we would have had to run MP3D with at least 125,000 particles to achieve realistic cache behavior. This would have taken extremely long to run.

We see no easy answer to the above question and are currently investigating the issues. For this study, however, we have chosen to scale down the cache sizes to get a more realistic problem size to cache size ratio. We scale down the processor caches to 2 Kbyte primary and 4 Kbyte secondary caches.[4] For MP3D, we get miss ratios approximating a large problem with full-size caches. However, we use only 10,000 particles and thus reduce the simulation time substantially. The data sets for the other two applications were also adjusted to get realistic cache hit ratios and reasonable run times. For LU, the data set size is chosen such that the data starts fitting into the combined caches of the processors only when the bottom third of the matrix remains to be factored. As a result, the processors get poor cache hit ratio in the beginning, and high hit ratios towards the end. This kind of behavior is not atypical of many numerical applications. For PTHOR, our experiments use a circuit with 11,000 gates. However, on the real machine, we expect to be using circuits with hundreds of thousands of gates. We thus reduce the cache size and the circuit size proportionately. To substantiate our results, we have also done experiments with larger cache sizes. Although we do not present the results here, due to lack of space, the results showed similar trends.

3 Coherent Caches

The first of the four techniques that we study is caching of shared data. The use of processor caches is a well accepted technique for reducing latencies in uniprocessors. Their use in multiprocessors, however, is complicated by the fact that the caches need to be kept coherent. While the coherence problem is easily solved for small bus-based multiprocessors through the use of snoopy cache-coherence protocols [4], the problem is much more complicated for large-scale multiprocessors that use general interconnection networks [5]. As a result, some existing large-scale multiprocessors do not provide caches (e.g., BBN Butterfly [26]), others provide caches that must be kept coherent by software (e.g., IBM RP3 [23]), and still others provide full hardware support for coherent caches (e.g., Stanford DASH [19]). In this section we evaluate the performance benefits when both private and shared read-write data are cacheable as allowed by hardware coherent caches versus the case when only private data are cacheable.

An alternative to hardware coherence is software cache coherence. Software schemes require sophisticated compiler technology and, in general, are conservative since they do not employ full dynamic information. This implies that the performance of software coherence schemes will usually lie between not caching shared data and hardware coherent caching of shared data. Due to lack of appropriate compiler technology, we could not evaluate the effectiveness of software schemes.

Figure 2 presents a breakdown of the normalized execution times with and without caching of shared data for each of the applications. Private data are cached in both cases. The experiments assume the sequential consistency model, so that no buffering or pipelining of cache misses is allowed. The execution time of each application is normalized to the execution time of the case where shared data

[3]The main memory for storing columns that are owned by a processor is allocated from shared-memory in that processor's node.

[4]These caches are only used for shared data. Instruction and private data references are not sent to the cache simulator and are implicitly assumed to hit in the cache.

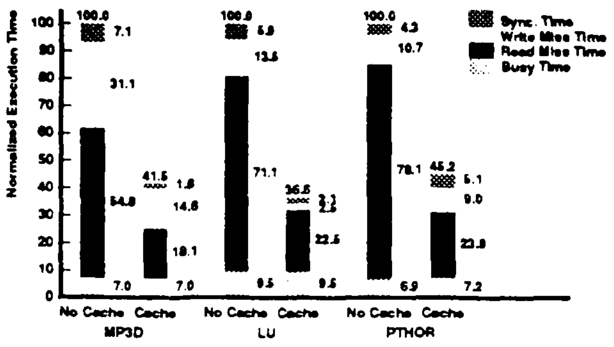

Figure 2: Effect of caching shared data.

is not cached. The bottom section of each bar represents the busy time or useful cycles executed by the processor. The section above it represents the time that the processor is stalled waiting for reads. The section above that is the amount of time the processor is stalled waiting for writes to be completed. The top section, labeled synchronization time, accounts for the time the processor is stalled due to locks and barriers.

As expected, the caching of shared read-write data provides substantial gains in performance, with benefits ranging from 2.2 to 2.7 fold improvement for the three programs. The largest benefit comes from the reduction in cycles wasted due to read misses. The cycles wasted due to write misses are also reduced, although the magnitude of the benefits varies across the three programs due to different write hit rates. The cache hit rates achieved by MP3D, LU, and PTHOR are 80%, 66%, and 77% respectively for shared-read references, and 75%, 97%, and 47% for shared-write references. It is interesting to note that these hit rates are substantially lower than the usual uniprocessor hit rates. The low hit rates arise from several factors: the data set size for engineering applications is large, parallelism decreases spatial locality in the application, and communication among processors results in invalidation misses. Still, hardware cache coherence is an effective technique for substantially increasing the performance with no assistance from the compiler or programmer.

Although caching shared data improves the performance substantially, the large number of cache misses and the large latency of each miss still keep the processor utilizations low (about 17% for MP3D, 26% for LU, and 16% for PTHOR). The next three sections study the effect of three different techniques for dealing with the large latency of cache misses by overlapping them with other computation and memory accesses. We assume hardware coherent caches for the rest of this study.

4 Relaxing the Memory Consistency Model

One way to remedy the large latency of cache misses is to hide the latency of accesses by buffering and pipelining the misses. Unfortunately, as a result of the combination of distributed memory, caches, and general interconnection networks used by large-scale multiprocessors [3, 19, 23], multiple requests issued by a processor may execute out of order. This may result in incorrect program behavior if the program depends on certain accesses to complete in order. Consequently, restrictions have to be placed on the types of buffering and pipelining allowed. These restrictions are determined by the memory consistency model supported by the multiprocessor.

Several memory consistency models have been proposed. The strictest model is that of *sequential consistency* [16] (SC). It requires the execution of a parallel program to appear as some interleaving of the execution of the parallel processes on a sequential machine. Unfortunately, SC imposes severe restrictions on the outstanding accesses that a process may have, thus limiting the buffering and pipelining allowed. One of the most relaxed models is the *release consistency* [9] (RC) model. Release consistency requires that synchronization accesses in the program be identified and classified as either *acquires* (e.g., lock) or *releases* (e.g., unlock). An acquire is a read operation (can be part of a read-modify-write) that gains permission to access a set of data, while a release is a write operation that gives away such permission. This information is used to provide flexibility in buffering and pipelining of accesses between synchronizations. The main advantage of the relaxed models is the potential for increased performance. The main disadvantage is increased hardware complexity and a more complex programming model.

Other relaxed models that have been discussed in the literature are *processor consistency* [9, 11], *weak consistency* [6], and *DRF0* [1]. These models fall between sequential and release consistency models in terms of flexibility and are not considered further in this study. For a detailed performance evaluation of relaxed memory consistency models, we refer the reader to a previous study [8].

4.1 Implementation of Consistency Schemes

Sequential consistency is satisfied in our implementation by ensuring that the memory accesses from each process complete in the order that they appear in the program. This is achieved by delaying the issue of an access until the previous access completes. The processors used in this study already stall on reads until the read access is satisfied. In addition, under SC, we explicitly stall the processor after every write until the write completes.

Release consistency can be satisfied by (i) stalling the processor on an acquire access until it completes and (ii) delaying the completion of a release access until all previous memory accesses complete. In the implementation assumed in this paper, the first condition is automatically satisfied because the processor stalls on all read accesses (including acquires) until the read is complete. To satisfy the second condition for RC, the write buffer is stalled on a release access until previously issued writes complete. To fully realize the benefits of RC, we allow reads to bypass the write buffer and provide a lockup-free cache such that reads can be serviced while there are write misses outstanding [8]. This ensures that reads are not stalled due to previous writes. The lockup-free cache also allows multiple write accesses to be pipelined.

Although the conditions for satisfying RC allow accesses and computation following a read to be overlapped and pipelined with the read, the implementation we study does not allow such overlap since reads are blocking. The design of processors that allow multiple outstanding reads and out-of-order execution of instructions is a current topic of research. However, the feasibility of such processors in addition to their effectiveness in hiding the latency of reads is still an open question.

The cost of implementing RC over SC arises from the extra hardware cost of providing a lockup-free cache and keeping track of multiple outstanding requests. Although this cost is not negligible, the same hardware features are also required to support prefetching and multiple contexts.

4.2 Comparison of SC versus RC

Figure 3 presents the breakdown of execution times under SC and RC for the three applications. Some general observations that can be made from the breakdown are the following: (i) the major reason for RC outperforming SC is that RC does not stall the processor on write accesses; and (ii) the read miss time forms a large portion of the idle time, especially once we move from SC to RC. As can be seen from the results, RC removes all idle time due to write miss

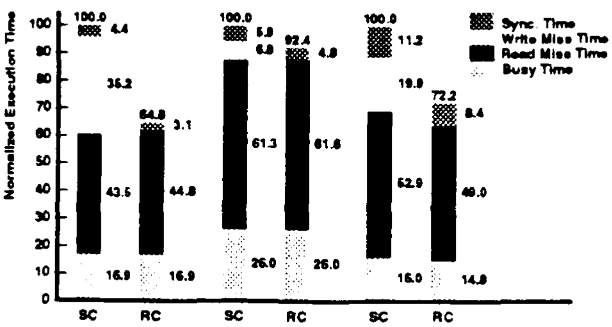

Figure 3: Effect of relaxing the consistency model.

latency. The gains are large in MP3D and PTHOR since the write-miss time constitutes a large portion of the execution time under SC (35% and 20%, respectively), while the gain is small in LU due to the relatively small write-miss time under SC (7%).

The pipelining of writes under RC provides another way in which RC can outperform SC. If there is a release operation (e.g., unlock) behind several writes in the write buffer, then a remote processor trying to do an acquire (e.g., lock on the same variable) can observe the release sooner, thus spinning for a shorter amount of time. Indeed, Figure 3 shows that synchronization times do decrease under RC. Overall, the release consistency model provides a speedup over sequential consistency of about 1.5 for MP3D, 1.1 for LU, and 1.4 for PTHOR.

While relaxing the memory consistency model effectively hides the latency of write accesses, the latency of read misses still remains. This is partly due to the fact that a processor with blocking reads does not allow a read miss to be overlapped with future computation and memory accesses. In light of the fact that read miss times constitute a large portion of the execution time (especially when the write miss time is removed), there is still room for large performance gains for techniques that can hide this latency. Indeed, the prefetching and multiple context techniques discussed in the next two sections attain most of their benefit by tackling the latency of reads.

5 Prefetching

Although release consistency hides much of the latency of write misses through buffering and pipelining, it still suffers during read misses when reads are blocking. These remaining misses can often be anticipated through knowledge of an application's reference behavior. Prefetching uses this knowledge to move data close to the processor before it is actually needed.

Prefetching can be classified based on whether it is *binding* or *non-binding*, and whether it is controlled by *hardware* or *software*. With binding prefetching, the value of a later reference (e.g., a register load) is bound at the time when the prefetch completes. This places restrictions on when a binding prefetch can be issued, since the value will become stale if another processor modifies the same location during the interval between prefetch and reference. Binding prefetching studies done by Lee [18] reported significant performance loss due to such limitations. In contrast, with non-binding prefetching the data is brought close to the processor, but remains visible to the cache coherence protocol to keep it consistent until the proces[illegible] read[illegible]. Hardware-controlled prefetching includes schemes such as long cache lines and instruction look-ahead [17]. The effectiveness of long cache lines is limited by the reduced spatial locality in multiprocessor applications [7, 29], while instruction look-ahead is limited by branches and the finite look-ahead buffer size. With software-controlled prefetching, explicit prefetch instructions are issued. Software control allows the prefetching to be done selectively (thus reducing bandwidth requirements) and extends the possible interval between prefetch issue and actual reference, which is very important when latencies are large. The disadvantages of software control include the extra instruction overhead to generate the prefetches as well as the need for sophisticated software intervention. In this study, we consider *non-binding software-controlled prefetching* [22].

The benefits due to prefetching come from several sources. The most obvious benefit occurs when a prefetch is issued early enough that the line is already in the cache by the time it is referenced. However, prefetching can improve performance even when this is not possible. When multiple prefetches are issued back-to-back, the latency of all but the first prefetched reference can be hidden due to the pipelining of the memory accesses. Prefetching offers another benefit in multiprocessors that use an ownership-based cache coherence protocol [4]. If a line is to be modified, prefetching it directly with ownership can significantly reduce the write latencies and the ensuing network traffic for obtaining ownership. Network traffic is reduced in read-modify-write situations, since prefetching with ownership avoids first fetching a read-shared copy.

5.1 Prefetching Implementation and Assumptions

In our model, a prefetch instruction is similar to a write in that it is issued to a prefetch buffer (which is identical to a write buffer, except that it only handles prefetch requests) and does not block the processor. The reason for having a separate prefetch buffer is to avoid delaying prefetch requests unnecessarily behind writes in the write buffer [22]. We model a prefetch buffer that is 16 entries deep. Once the prefetch reaches the head of the prefetch buffer, the secondary cache is checked to see whether the line is already present. If so, the prefetch is discarded. Otherwise the prefetch is issued onto the bus, where it is treated like any normal memory request. When the prefetch response returns to the processor, it is placed in both the secondary and primary caches. If the processor is executing when this cache fill begins, it is stalled for four cycles (since the cache line size is four words) to model the effect that no loads or stores can be executed while the cache is busy. If a processor references a location it has prefetched before the result has returned, the reference request is combined with the prefetch request so that a duplicate set of messages is not sent out and so that the reference completes as soon as the prefetch result returns.

Since we did not want to be constrained by the limits of existing compiler technology to automatically add prefetching, and because such a compiler was not available to us, prefetches were introduced manually at the source level of each application through macro statements. These macros covered both *read* and *read-exclusive* prefetches, as well as single cache line and block prefetches. A read prefetch brings data into the cache in a read-shared mode, while a read-exclusive prefetch also acquires exclusive ownership of the line, enabling a write to that location to complete quickly.

5.2 Prefetching Results

We begin with a description of how prefetching was inserted into each application, and then discuss the results for both sequential and release consistency.

MP3D: Most of the time is spent in a loop where each processor takes a particle and moves it through one time step. The overwhelming majority of cache misses are caused by references to two structures within this loop: (i) the particle which is being moved (34% of misses), and (ii) the space cell where the particle resides (50%). Particles are statically assigned to processors and are allocated to the corresponding local memories,

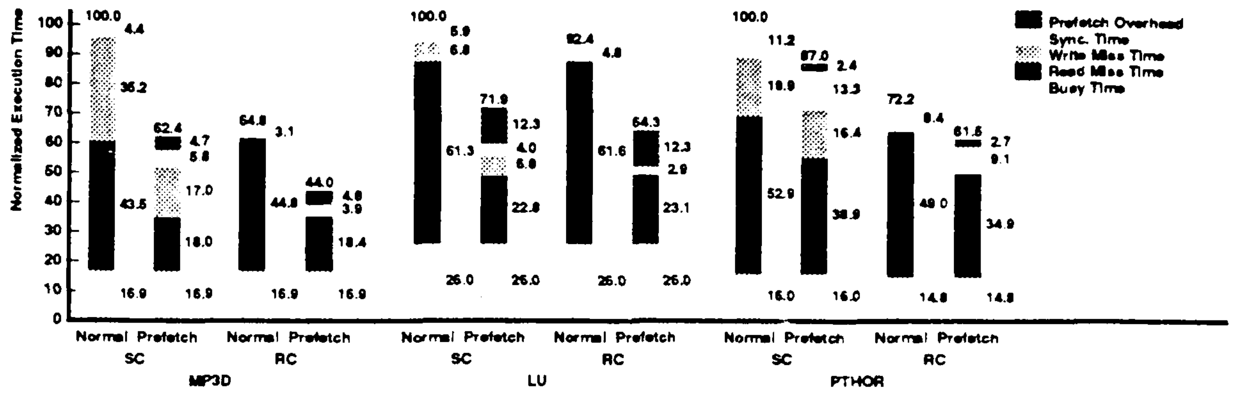

Figure 4: Effect of prefetching.

while space cells are uniformly allocated since they are shared among processors.

Since a particle must be referenced to determine the space cell it occupies, we prefetch a particle record two iterations before its turn to be moved. In the iteration following the prefetch, the particle is read, and the associated space cell is determined and prefetched. As a result, when it is time for the particle to be moved, both the particle and space cell records are available in the cache. We also prefetch several other references that occur at time step boundaries, such that a total of 87% of all misses are prefetched (we will refer to this as the *coverage factor*). Read-exclusive prefetches are used since the objects are modified during each iteration.

LU: The matrix columns are statically assigned to the processors in an interleaved manner, and are allocated to the corresponding local memories. The main computation done by each processor consists of reading a pivot column once it is produced, and applying the pivot column to each column to its right that the processor owns. There are three primary sources of misses in LU: (i) the pivot column when it is read for the first time (8%); (ii) the pivot column when it is replaced by a column it is applied to and needs to be refetched (17%); and (iii) the owned columns that the pivot column is applied to (64%). This last set of misses occurs because the combined size of the owned columns is larger than the size of the cache.

Each time the pivot column is applied to an owned column, we prefetch the pivot column in read-shared mode and the owned column in read-exclusive mode. Although prefetching the pivot column each time causes redundant prefetches, it reduces the misses when the pivot column is replaced from the processor's cache, resulting in a total coverage factor of 89%. We found that it is better to evenly distribute the issue of prefetches throughout the computation rather than prefetching an entire column in a single burst, in order to avoid hot-spotting problems.

PTHOR: In the main computational loop, each processor picks up an activated logic element, computes any changes to the element's outputs, and schedules new input events for elements that are affected by the changes. One of the main data structures in the program is the *element record*, which stores all information about the type and state of the element. Several fields in the record are pointers to linked lists, or are pointers to arrays that in turn point to linked lists. Prefetching is complicated by the presence of linked lists, since to prefetch a list it is necessary to dereference each pointer along the way.

We first reorganized the element record and grouped entries based on whether they were likely to be modified, likely to be read but not modified, or likely not to be referenced. Whenever a processor picks an element from a task queue, we prefetch the element record entries accordingly. In addition, we prefetch the first several levels of the more important linked lists. Due to the complex control structure of the application, it is difficult to determine where the misses occur. Despite the aid of profiling markers that helped determine which sections of code were generating misses, we were only able to increase the coverage factor to 56%.

The results of the prefetching experiments are shown in Figure 4. Notice that a new section has been added to the execution time bar to account for prefetching overhead. This includes any extra instructions executed to do prefetching (e.g., evaluation of conditional statements that help decide whether to prefetch or not, instructions to do address computation, and the prefetch instruction itself), any time for which issuing a prefetch stalls the processor due to a full prefetch buffer, and any stall time due to the primary cache being filled with a prefetched line.

For sequential consistency we see that most of the benefit comes from reduced read latencies, and that this more than offsets the added prefetch overhead. While read-exclusive prefetching effectively reduces write latencies for MP3D, it offers little or no improvement for PTHOR (since only a small fraction of prefetches are read-exclusive) and LU (since write latencies are already small because owned columns are allocated to local memory). Prefetch overhead is substantial in the case of LU since there is very little computation between references, causing the prefetch generation instructions to be a large fraction of total instructions. The overhead due to primary cache fills is much less of a problem. The main difference we see when prefetching is combined with release consistency is that the write latency has already been eliminated, so the benefits come strictly through reduced read latency.

The benefits of prefetching are limited by several factors. First, inserting the prefetches can be difficult. This was especially true for PTHOR. The difficulty is both identifying the references that need to be prefetched and scheduling the prefetches far enough in advance to effectively hide latency. We are currently working on compiler technology to automate this process. Secondly, even if a reference is prefetched far enough in advance, cache interference may cause it to be knocked out of the cache before it can be referenced. This interference can be either self-interference in the form of replacements or external interference caused by invalidations. Finally, the overhead of adding prefetches can potentially offset much of the gain that is realized through reduced latencies, as we see in the case of LU.

The advantage of prefetching is that significant gains can be achieved by inserting only a handful of prefetches when the access patterns are regular and predictable. For MP3D, adding only

16 lines to the source code resulted in speedups of 1.60 and 1.47 under SC and RC, respectively. Another great advantage in terms of hardware cost is that prefetching can be implemented using existing commercial processors.

6 Multiple-Context Processors

Although prefetching is useful for many applications, it requires explicit programmer or compiler intervention. Processors with multiple hardware contexts [3, 13, 14, 27] do not have this disadvantage. They make use of increased concurrency to hide latency. Each processor has several processes assigned to it, which are kept as hardware contexts. When the context that is currently running encounters a long-latency operation, it is switched out and another context is started. In this manner the memory latency of one context can be hidden with computation of another context. Given processor caches, the interval between long-latency operations (i.e., cache misses) becomes fairly large, allowing just a handful of hardware contexts to hide most of the latency [2, 25, 30]. This is in contrast to the early multiple-context processors such as the HEP [27], where context switches occurred on every cycle.

The performance gain to be expected from multiple context processors depends on several factors. First, there is the number of contexts. With more contexts available, we are less likely to have a completely idle processor due to running out of ready-to-run contexts. On the other hand there might not be enough parallelism in the application to support many contexts per processor. Secondly, there is the context switch overhead. If the overhead is a sizeable fraction of the typical run lengths (time between misses) encountered, a lot of time will be wasted with the switching of contexts. Shorter context switch times require a more complex processor. Thirdly, the performance depends on the application behavior. Under ideal conditions where latencies are constant and misses occur at regular intervals, a multiple context processor can achieve a high utilization. However, with real applications, latencies can vary depending on where the data resides and what state it is in. At the same time misses may be clustered. Both of these will make it impossible to completely overlap computation of one context with memory accesses of the other contexts. Hence the processor utilization will not reach its full potential. Lastly, multiple contexts themselves will affect the performance of the memory subsystem. The different contexts share a single processor cache and can interfere with each other, both constructively and destructively. Also, just as is the case with relaxed consistency and prefetching, the memory system is more heavily loaded by multiple contexts, and thus latencies may increase.

We presented a preliminary investigation of multiple-context processors in a previous study [30]. More recently, there have also been two analytical evaluations of multiple contexts [2, 25]. In this study we present a more detailed simulation evaluation of the performance of multiple-context processors, and we also consider the combined effect with other latency-hiding techniques. We use processors with two and four contexts. We do not consider more contexts per processor because sixteen 4-context processors require 64 parallel threads and some of our applications do not get very good speedup beyond this point. We use two different context switch overheads: 4 and 16 cycles. A four-cycle context switch overhead corresponds to flushing/loading a typical RISC pipeline when switching to the new instruction stream. This type of processor would require multiple register sets to allow fast switching between them. An overhead of sixteen cycles corresponds to a less aggressive implementation. In our study, we include additional buffers to avoid thrashing when two contexts try to read two memory lines that map to the same cache line. Without the buffers, the two contexts could continually knock the other context's line out of the cache, causing a never-ending stream of read misses.

6.1 Results with Multiple-Context Processors

We start our investigation of multiple contexts with an evaluation of their benefit under sequential consistency. Later we will examine the combined benefit when the consistency model is relaxed and prefetching is added.

Refer to Figure 5 for the results under sequential consistency. We show results for single-context processors as well as 2- and 4-context processors with context switching penalties of 4 and 16 cycles. The height of each bar represents the execution time of the application under the given scheme. Each bar is broken down into the following components: *busy* time which represents actual work being done by the processor, *switching* time incurred when switching from one context to the next, *all idle* time which is the total time when all contexts are idle waiting for a reference to complete, and *no switch* time which represents time when the current context is idle but is not switched out. Most of the latter idle time is due to the fact that the processor is locked out of the primary cache while fill operations of other contexts complete. Under sequential consistency, some of the *no switch* idle time is due to write hits in the secondary cache, which stall the processor for two cycles.

MP3D benefits greatly from the use of multiple contexts (see the top of Figure 5). The median run lengths are about 11 cycles long, and the average miss latencies are 50–70 cycles long. With a context switch overhead of four cycles, we expect to need about 5 contexts to completely hide the miss latencies. With two contexts we see some reduction in *all idle* time and with four contexts an additional portion of this idle time is eliminated. However, with multiple contexts we now have additional idle time in the form of context switch overhead. This time is especially significant when the context switch overhead is 16 cycles. It is interesting to note that there is very little performance improvement going from a switch penalty of 16 cycles to one of 4 cycles with 2 contexts. The context switch time saved simply shows up as additional *all idle* time.

The behavior of LU (middle of Figure 5) is completely dominated by cache interference. With a single context, the read and write hit rates are 66% and 97% respectively. With two contexts they deteriorate to 56% and 38%, and with 4 contexts they are down to 50% and 16%. These additional misses lead to more context switches and more time wasted on context switching. With 16 cycle context switch overhead, performance gets worse as more contexts are added. Even though some of the latencies are hidden, the time wasted on context switches dominates. With the 4 cycle context switch overhead, some gains are possible. The median run lengths are 6 cycles long, and the average miss latencies are 20–27 cycles long. The miss latencies are low because a high proportion of them are due to the owned columns of the matrix, which are allocated from the local portion of shared memory.

PTHOR (bottom of Figure 5) shows another interesting effect. There is not enough parallelism available in the application to achieve good speedup with a large number of processors or contexts. So even though the run lengths and latencies are favorable (they are 7 and 60–80 cycles respectively), the gains achieved with two contexts are small. Four contexts actually do worse than two, no matter what the context switch overhead is. There simply is not enough parallelism available to provide useful work for four contexts per processor. The additional contexts spend most of their time busy-waiting on an empty task queue. During this time they hold up the useful work being done by the other contexts that did manage to find a task. The additional instruction cycles used for spinning on the task queue are reflected in the graphs as extra busy time. We note that when we run PTHOR with four processors instead of sixteen, we find that multiple contexts achieve much greater gains: four context-processors run about twice as fast as single-context processors.

The conclusion from the results of our experiments with multiple contexts under sequential consistency is that multiple contexts can

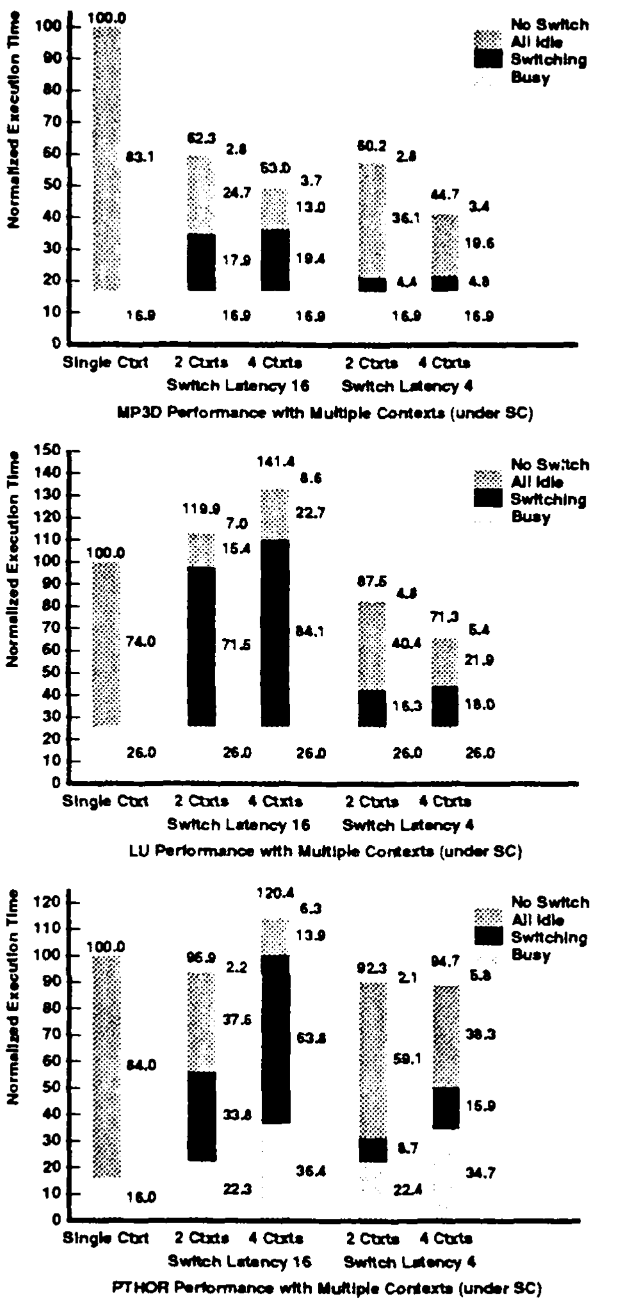

Figure 5: Effect of multiple contexts.

increase performance significantly when the run length to latency ratio is favorable. However, enough parallelism must be available in the application to keep the additional contexts busy. We also note that destructive interference of the contexts in the processor cache can undo any gains achieved. Interference is more of a problem with multiple contexts than with prefetching because multiple working sets interfere with each other in the same cache. The smaller the number of cycles required for context switching, the lower the total overhead due to multiple contexts. A context switch overhead of 16 cycles introduces significant overhead, whereas the overhead is much more reasonable with a 4-cycle switch penalty. The typical run lengths and latencies encountered suggest that a small number of contexts (such as 4) is sufficient to achieve most of the latency hiding benefits.

6.2 Effect of Combining other Schemes with Multiple Contexts

We have seen that multiple contexts with sequential consistency can increase performance substantially under favorable circumstances. An interesting question is whether multiple contexts can gain any extra performance when combined with relaxed consistency models. The left and middle sections of the graphs in Figure 6 show the performance of multiple contexts with SC and RC, respectively. We only show results for a context-switch overhead of 4 cycles. The major difference between release consistency and sequential consistency is that write misses are no longer considered long latency operations from the processor's perspective, since writes are simply put into the write buffer. We thus find that median run lengths between switches have increased (from 11 to 22 cycles for MP3D and from 6 to 14 cycles for LU) and that fewer contexts are required to eliminate most of the remaining read miss latencies. As a result, the gains achieved with four contexts over two contexts are also diminished. As is apparent from the results, there is some benefit from relaxing the consistency model with multiple contexts. For the 4-context case, performance improved by a factor of 1.32 for MP3D, 1.24 for LU, and 1.17 for PTHOR when going from SC to RC.

Finally, let us consider the combined effect of multiple contexts and prefetching (see the right portions of the graphs of Figure 6). In general, prefetching and multiple contexts aim to hide the same idle time—that caused by long latency read and write misses. We thus expect the gain of prefetching with multiple contexts to be less than with single contexts. This is indeed the case. Prefetching improves performance only in the cases where multiple contexts have not been able to hide most of the latency. For example, with MP3D under release consistency, there is a significant performance improvement when going from two to four contexts (top of Figure 6). A similar (albeit somewhat smaller) gain can be achieved by applying prefetching to the two-context case. However, combining prefetching with four contexts yields worse performance. Here we are paying the price of prefetching overhead, but are not reducing the latency. LU and PTHOR show similar trends.

Although prefetching and multiple contexts each aim to reduce the same latency, there are some distinguishing features. The big advantage of prefetching is that it does not require a special processor. Also, many more accesses can be outstanding at any given time, thus allowing their latencies to be overlapped. With prefetching, each processor can issue an essentially unlimited number of prefetch requests. Multiple contexts, on the other hand, are limited by the total number of contexts, which is expected to be a small number. The advantage of multiple contexts is that they can handle very irregular access patterns which cannot be prefetched efficiently. In addition, multiple contexts do not require software support.

In our study, prefetching was added without any regard to the effect it might have on multiple contexts. For example, prefetches are added in the single context case even if they cannot be issued early enough to completely hide the latency. With multiple contexts, the benefit of such partial latency hiding is diminished because a miss will occur, triggering a context switch. If the multiple contexts would have hidden the miss latency anyway, prefetch overhead has just been added without any benefit. This effect becomes more significant as the number of contexts is increased. Multiple contexts thus add another interesting dimension to the question of when to prefetch.

In summary, release consistency helps multiple contexts because it eliminates writes as long latency operations, thus increasing run lengths and allowing the remaining latency to be hidden with fewer contexts. The benefit of adding prefetching to multiple contexts is small, and may even be negative, especially when little latency is left to hide. Inserting prefetches with more awareness of their effect on the performance of multiple contexts may achieve better results.

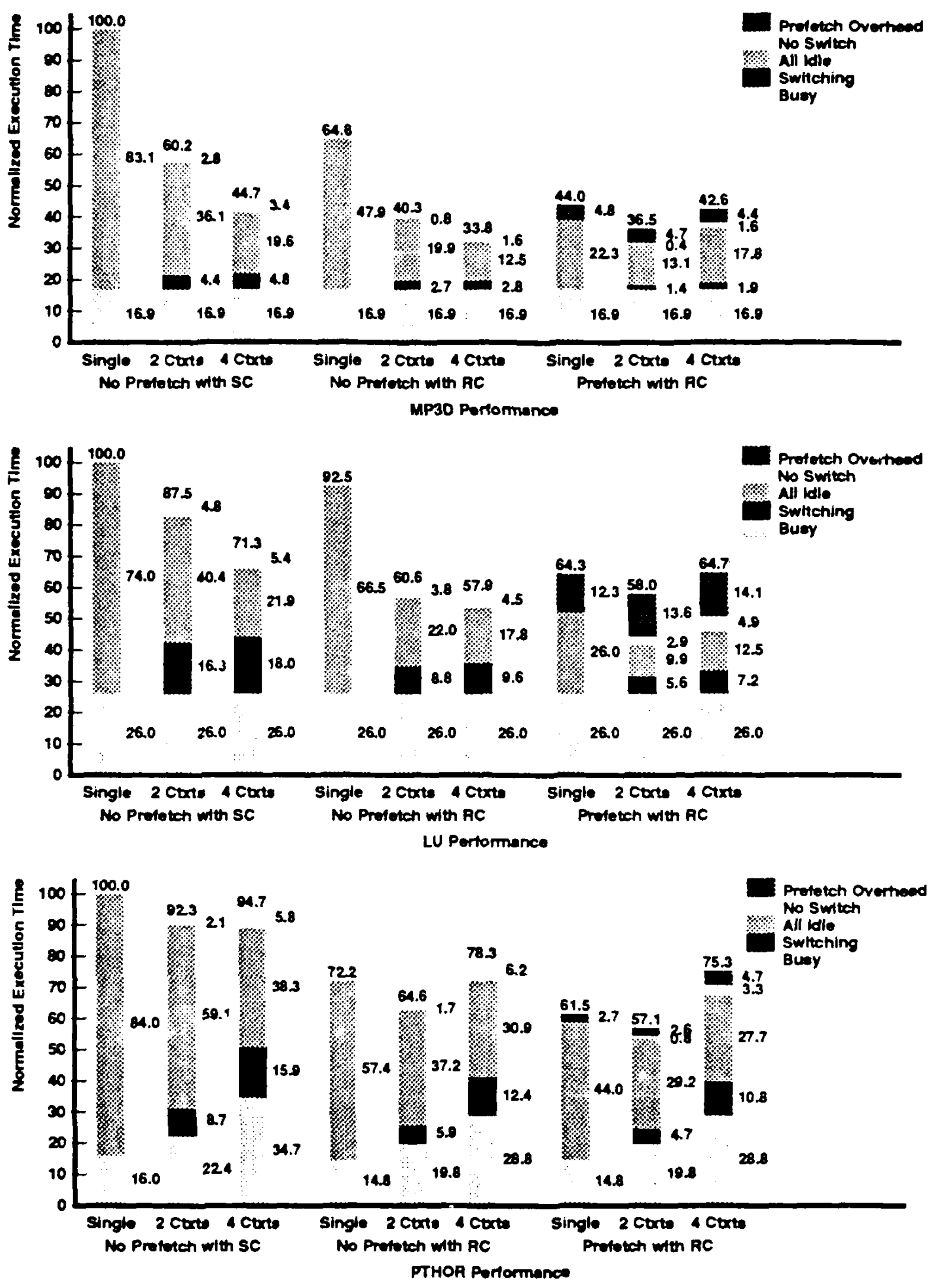

Figure 6: Effect of combining the schemes (multiple-context schemes have a 4-cycle switch latency).

7 Concluding Remarks

While several latency hiding techniques have been proposed in the past, a study evaluating the relative performance benefits of these techniques and their combinations had been lacking. In this paper, we have presented such an evaluation for four techniques—coherent caches, relaxed memory consistency, prefetching, and multiple contexts—[illegible] architectural assumptions and benchmarks. As expected, the largest single improvement in runtime, a factor of 2.2 to 2.7, came from coherent caches. Relaxing the consistency model provided additional performance gains of 1.1 to 1.5, arising mainly from the hiding of write latencies. Similar to the gain from caches, this gain is automatic as long as programs use explicit synchronization. Since the relaxed models hide latencies by allowing multiple outstanding references, the main hardware requirement (in addition to coherent caches) is lock-up free caches. Lock-up free caches are also necessary for prefetching and for multiple-context processors, and thus form a universal requirement for latency hiding techniques.

As intended, prefetching was very successful in reducing the stalls due to read latencies (factor of 2.4 for MP3D, 2.7 for LU, and 1.4 for PTHOR). Prefetching was less effective in reducing write latency under the strict consistency model, but combined well with the relaxed consistency model to eliminate both types of latency. The overall speedups were 2.3 for MP3D, 1.6 for LU, and 1.6 for PTHOR. While prefetching has the drawback that it requires compiler or programmer intervention, a significant advantage is that it requires no major hardware support beyond that needed by RC, and it can easily be incorporated into systems built using existing commercial microprocessors.

The multiple context approach, while requiring significant hardware support, provided mixed results when the context-switch overhead was 16 cycles. In cases where the concurrency was low (e.g., PTHOR) or where there was substantial cache interference (e.g., LU), the use of multiple contexts made the performance worse. The use of relaxed consistency helped multiple context performance by hiding write latencies and increasing the run lengths. Under an aggressive implementation, with use of 4 contexts and a context-switch overhead of 4 cycles, the performance benefits were a factor of 3.0 for MP3D, 1.7 for LU, and 1.3 for PTHOR. The interaction of multiple contexts with prefetching was shown to be complex. Oftentimes the performance became worse when the two were combined together, because the prefetch overheads were greater than the additional latency that was hidden. To achieve better results, it appears that the prefetching strategy must become more sensitive to the presence of multiple contexts.

8 Acknowledgments

We thank the reviewers for their comments. This research was supported by DARPA contract N00014-87-K-0828. Anoop Gupta is partially supported by a NSF Presidential Young Investigator Award with matching funds from Sumitomo, Tandem, and TRW. Wolf-Dietrich Weber is partially supported by IBM and Kourosh Gharachorloo is partially supported by Texas Instruments.

References

[1] S. Adve and M. Hill. Weak ordering - A new definition. In *Proc. Int. Symp. Comput. Arch.*, pages 2–14, May 1990.

[2] A. Agarwal. Performance tradeoffs in multithreaded processors. MIT VLSI Memo 89-566, Lab. for Comput. Sci., Submitted for publication, September 1989.

[3] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *Proc. Int. Symp. Comput. Arch.*, pages 104–114, May 1990.

[4] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 4(4):273–298, 1986.

[5] W. J. Dally. Wire efficient VLSI multiprocessor communication networks. In *Stanford Conference on Advanced Research in VLSI*, 1987.

[6] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proc. Int. Symp. Comput. Arch.*, pages 434–442, June 1986.

[7] S. J. Eggers and R. H. Katz. Evaluating the performance of four snooping cache coherency protocols. In *Proc. Int. Symp. Comput. Arch.*, pages 2–15, May 1989.

[8] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Int. Conf. Arch. Support Prog. Lang. Oper. Syst.*, April 1991.

[9] K. Gharachorloo D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. Int. Symp. Comput. Arch.*, pages 15–26, May 1990.

[10] S. R. Goldschmidt and H. Davis. Tango introduction and tutorial. Technical Report CSL-TR-90-410, Stanford University, 1990.

[11] J. R. Goodman. Cache consistency and sequential consistency. Technical Report no. 61, SCI Committee, March 1989.

[12] E. Gornish, E. Granston and A. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Int. Conf. Supercomputing*, 1990.

[13] R. H. Halstead, Jr. and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proc. Int. Symp. Comput. Arch.*, pages 443–451, June 1988.

[14] R. A. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *Proc. Int. Symp. Comput. Arch.*, pages 131–140, June 1988.

[15] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proc. Int. Symp. Comput. Arch.*, 1981.

[16] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C-28(9):241–248, September 1979.

[17] R. L. Lee. *The Effectiveness of Caches and Data Prefetch Buffers in Large-Scale Shared Memory Multiprocessors.* PhD thesis, University of Illinois at Urbana-Champaign, May 1987.

[18] R. L. Lee, P.-C. Yew, and D. H. Lawrie. Data prefetching in shared memory multiprocessors. In *Proc. Int. Conf. Paral. Proc.*, pages 28–31, August 1987.

[19] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proc. Int. Symp. Comput. Arch.*, May 1990.

[20] E. Lusk, R. Overbeek, et al. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, Inc., 1987.

[21] J. D. McDonald and D. Baganoff. Vectorization of a particle simulation method for hypersonic rarified flow. In *AIAA Thermodynamics, Plasmadynamics and Lasers Conference*, June 1988.

[22] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *J. Paral. Dist. Computing*, to appear in June 1991.

[23] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proc. Int. Conf. Paral. Proc.*, pages 764–771, 1985.

[24] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Rice University, May 1989.

[25] R. H. Saavedra-Barrera, D. E. Culler, and T. von Eicken. Analysis of multithreaded architectures for parallel computing. In *ACM Symp. Paral. Alg. Arch.*, July 1990.

[26] G. E. Schmidt. The Butterfly parallel processor. In *Proc. Int. Conf. Supercomputing*, pages 362–365, 1987.

[27] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. *SPIE*, 298:241–248, 1981.

[28] L. Soule and A. Gupta. Parallel distributed-time logic simulation. *IEEE Design and Test of Computers*, 6(6):32–48, December 1989.

[29] J. Torrellas, M. S. Lam, and J. L. Hennessy. Measurement, analysis, and improvement of the cache behavior of shared data in cache coherent multiprocessors. Technical Report CSL-TR-90-412, Stanford University, Feb. 1990.

[30] W.-D. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *Proc. Int. Symp. Comput. Arch.*, pages 273–280, June 1989.

[31] A. W. Wilson, Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proc. Int. Symp. Comput. Arch.*, pages 244–252, June 1987.